Lecture 12: Agile/Lean Methods

Dietmar Pfahl
email: dietmar.pfahl@ut.ee
Schedule of Lectures

Week 01: Introduction to SE
Week 02: Requirements Engineering I
Week 03: Requirements Engineering II
Week 04: Analysis
Week 05: Development Infrastructure
Week 06: Continuous Development and Integration
Week 07: Project Estimation / Architecture and Design I
Week 08: Architecture and Design II
Week 09: Verification and Validation I
Week 10: Verification and Validation II
Week 11: Refactoring (and TDD)
Week 12: Agile/Lean Methods
Week 13: Industry Guest Lecture – Anton Keks, Codeborne
Week 14: Course wrap-up, review and exam preparation
Week 15: No regular lecture – exam consultation on request
Industry Guest Lecture

- Date/Time:
  - Friday, 26-Nov, 10:15-12:00

- Topic: “Software Craftsmanship, the Codeborne Way”
- Speaker: Anton Keks, founder and co-owner of …
Industry Guest Lecture

- Date/Time:
  - Friday, 26-Nov, 10:15-12:00

- Topic: “Software Craftsmanship, the Codeborne Way”

- Speaker: Anton Keks, founder and co-

I will provide a Zoom link for this special session!
Exam → in Moodle!

• Despite the unpredictable situation regarding COVID-19, the exam will be delivered online via Moodle **BUT** in Delta Building
  - Stay tuned for further announcements!
• You must register in SIS for either Exam 1 or Exam 2 [or==xor]
• Only registered students will be able to access the exam in Moodle on the day for which they registered
• Exam Dates/Times:
  - Exam 1: Friday, 07-Jan-2021 at 10:15-12:45 - room 1037. (limit: 100 students)
  - Exam 2: Tuesday, 11-Jan-2021 at 14:15-16:45 - room 1037. (limit: 100 students)
Structure of Lecture 12

- Light-weight processes / Evolutionary development
- Agile Processes/Methods
  - Extreme Programming (XP)
  - Scrum (intro)
- KANBAN
- Lean Processes/Methods
Waterfall Process = Heavy-Weight
Waterfall Process = Heavy-Weight

Challenges:
- Changing Requirements
- Fixed-price/fixed-scope/fixed-deadline projects
- Heavy-weight process models (prescriptive)
- Taylorism, trying to create many specialised roles
From Heavy-Weight to Light-Weight

Functionality

Time

Requirements

Design

Implementation

Testing

Waterfall

Incremental

Iterative/Evolutionary
The Agile Manifesto

Kent Beck et al. (2001):

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.
There exists more than one Agile Method

- Crystal Clear
- DSDM
- FDD
- ... (omitted)
- AUP
- Scrum
- XP

AUP = Agile Unified Process
DSDM = Dynamic Systems Development Method
FDD = Feature-Driven Development
XP = Extreme Programming

LEAN

KANBAN
Process Frameworks Used in Estonia

1. Scrum
2. DevOps (CI/CD)
3. Iterative Development
4. Kanban
5. XP
## Process Frameworks Used in Sweden

<table>
<thead>
<tr>
<th>Frameworks/Methods</th>
<th>Often Used</th>
<th>Always Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Scrum</td>
<td>46%</td>
<td>8%</td>
</tr>
<tr>
<td>2. Iterative Development</td>
<td>48%</td>
<td>15%</td>
</tr>
<tr>
<td>3. Kanban</td>
<td>46%</td>
<td>3%</td>
</tr>
<tr>
<td>4. Classic Waterfall</td>
<td>38%</td>
<td>0%</td>
</tr>
<tr>
<td>5. XP</td>
<td>8%</td>
<td>0%</td>
</tr>
</tbody>
</table>

### Notes:
- **Do not know the framework**
- **We never use it**
- **We do not know if we use it**
- **We rarely use it**
- **We sometimes use it**
- **We often use it**
- **We always use the framework**
Structure of Lecture 11

- Light-weight processes / Evolutionary development
- Agile Processes/Methods
  - Extreme Programming (XP)
  - Scrum (intro)
- KANBAN
- Lean Processes/Methods
13 XP Practices

Project Cycle
- Planning Poker
- Small Releases
- Whole Team
- Customer Tests

Development Cycle
- Simple Design
- Pair Programming
- TDD (Unit Test)
- Refactoring

Supporting Practices
- Coding Standard
- Sustainable Pace (40-hour week)
- Metaphor (Common Understanding)
- Continuous Integration
- Collective Ownership
Requirements vs. User Stories

Traditional requirement – “shall” statements:

- “The system shall provide a user configurable interface for all user and system manager functions”
- “The user interface shall be configurable in the areas of:
  - Screen layout
  - Font
  - Background and text color

Corresponding “User Story”:

- “As a system user or system manager, …
- … I want be able to configure the user interface for screen layout, font, background color, and text color, …
- … So that I can use the system in the most efficient manner”
Planning Poker

Participants in planning poker include all of the developers on the team

- **Step 1:** Give each estimator a deck of cards
- **Step 2:** Moderator reads description of User Story to be estimated.
- **Step 3:** Product owner answers any question the estimators may have about the User Story.
- **Step 4:** Each estimator privately selects a card representing his or her estimate. Cards are not shown until each estimator has made a selection.

…
Planning Poker (cont’d)

- **Step 5:** When everyone has made an estimate, the cards are simultaneously turned over.

- **Step 6:** If estimates differ, the highest and lowest estimates are explained by the estimators - otherwise the estimation is completed for this User Story.

- **Step 7:** The group can discuss the story and their estimates for a few more minutes. The moderator can take any notes he/she thinks will be helpful when this story is being programmed and tested. After the discussion, each estimator re-estimates by selecting a card.

  -> Go to Step 5.

Note: In many cases, the estimates will already converge by the second round. But if they have not, continue to repeat the process. The goal is for the estimators to converge on a single estimate that can be used for the story. It rarely takes more than three rounds, but continue the process as long as estimates are moving closer together.
Simple Design

Characterisation:

- Four characteristics of simple design, listed in priority order:
  1. The system runs all the tests.
  2. It contains no duplicate code.
  3. The code states the programmers' intent very clearly.
  4. It contains the fewest possible number of classes and methods.

Guidelines to help in arriving at a simple design:

- Look for a simple – but not stupid – way to solve a problem. Pay attention to good design principles when forming a system incrementally. (-> design patterns)

- Don’t add infrastructure or other features that might be needed later. Chances are they won’t be (YAGNI: You Aren't Going to Need It). Let the user stories force you to change the design.

- Don't generalize a solution until it is needed in at least two places. Follow the first rule above and keep implementation simple. Let the second user pay for the generality.

- Seek out and destroy duplication and other ‘code smells’ (or: ‘design smells’). The practice of refactoring is the most powerful tool in the arsenal. It is through removing duplication that new classes, methods, and larger scale systems are born.

- Remember that it is just code. If it is getting overly complex and painful, delete it. It can always be recreated again in less time and better than the first time by leveraging what was learned the first time.

TDD / BDD

Refactoring
Test-Driven Development (TDD)

- Unit Test
- Functionality-oriented
- Regression testing can be automated

namespace UnitTestingExamples.Tests
{
    using System;
    using NUnit.Framework;
    [TestFixture]
    public class BankAccountTests
    {
        [Test]
        public void TestDeposit()
        {
            BankAccount account = new BankAccount();
            account.Deposit(125.0);
            account.Deposit(25.0);
            Assertion.AreEqual(150.0, account.Balance);
        }
    }
}

namespace UnitTestingExamples.Library
{
    using System;
    public class BankAccount
    {
        private double _balance = 0.0;
        public void Deposit(double amount)
        {
            _balance += amount;
        }
        public double Balance
        {
            get { return _balance; }
        }
    }
}
TDD & Refactoring Integrated

User Story (new functionality)

- Write a failing test.
- Clean up code.
- Make the test pass.
Test-Driven Development

Developer TDD => Unit Tests

Acceptance TDD => Acceptance Tests
also called: Behavior-driven development/testing (BDD)
Refactoring

• Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. (Invented by Martin Fowler)

• Many refactorings can be automated

• Catalogue of refactorings:
  
  http://www.refactoring.com/catalog/index.htm

• Note: It is not always clear
  
  • (a) how to detect refactoring opportunities and
  • (b) what refactoring(s) are most appropriate
  
Pair Programming

Characterisation:
• Two programmers work together at one computer
  • The driver, writes code
  • The observer (or navigator), reviews each line of code as it is typed in
• The two programmers switch roles frequently
Effectiveness and Efficiency of Pair Programming


Good?
Bad?
Overall Effect of Pair Programming

Finding 1:
PP is expensive and has only little positive effect on quality
Effect of Pair Programming for Junior Developers

Experience level = junior
All task types (easy and complex)

Finding 2: PP is expensive but seems to have a positive effect on quality

Effect of PP for Juniors

<table>
<thead>
<tr>
<th>Difference from individuals</th>
<th>Duration</th>
<th>Effort</th>
<th>Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 %</td>
<td>111 %</td>
<td>73 %</td>
</tr>
</tbody>
</table>

Percentage differences:
- Duration: 5%
- Effort: 111%
- Correctness: 73%
Effect of PP for **Juniors** per task complexity

- Experience level = junior
- Task types = easy | complex

**Finding 3:**
PP is expensive but seems to have a very positive effect on quality if task is complex
Effect of PP for **Seniors** per task complexity

Experience level = senior

Task types = easy | complex

Finding 4:
PP is expensive and has only little positive effect on quality – no matter whether task is easy or complex
So, when should we use Pair Programming?

<table>
<thead>
<tr>
<th>Programmer Expertise</th>
<th>Task Complexity</th>
<th>Use PP?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Junior</td>
<td>Easy</td>
<td>Yes</td>
<td>Provided that increased quality is the main goal</td>
</tr>
<tr>
<td></td>
<td>Complex</td>
<td>Yes</td>
<td>Provided that increased quality is the main goal</td>
</tr>
<tr>
<td>Intermediate</td>
<td>Easy</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Complex</td>
<td>Yes</td>
<td>Provided that increased quality is the main goal</td>
</tr>
<tr>
<td>Expert</td>
<td>Easy</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Complex</td>
<td>No</td>
<td>Unless you are sure that the task is too complex to be solved satisfactorily even by solo seniors</td>
</tr>
</tbody>
</table>

The question of whether PP is best, or not, is meaningless!
One should ask: In which situation is PP best to achieve a defined goal?

Importance of Context:
Helps construct/refine theory about when and how to do 'Pair Programming'
PP is NOT a Silver Bullet!
Pair Programming

Characterisation:
- Two programmers work together at one computer
  - The driver, writes code
  - The observer (or navigator), reviews each line of code as it is typed in
- The two programmers switch roles frequently

Challenges:
- Total effort (person-hours) increases
  - Management needs to balance faster completion of the work and reduced testing and debugging time against the higher cost of coding
  - Greatest benefit on tasks that the programmers do not fully understand before they begin: that is, challenging tasks that call for creativity and skill
  - On simple tasks, which the pair already fully understands, pairing results in a net drop of productivity.
- Productivity might drop when novice-novice pairing is used without coaching.

Benefits:
- Studies found that programmers working in pairs produce
  - shorter programs,
  - with better designs
  - and fewer bugs faster
Structure of Lecture 11

- Light-weight processes / Evolutionary development
- Agile Processes/Methods
  - Extreme Programming (XP)
  - Scrum (intro)
- KANBAN
- Lean Processes/Methods
The Term “Scrum”

Originates from Rugby
Meaning “crowded”
Complex move that requires team work
Scrum Elements – Process, Artifacts, Roles

http://www.scrumforteamsystem.com/processguidance/v1/Scrum/Scrum.html
What is Scrum?

• Agile Management Framework for SW development projects

• With a few clear rules:
  • Roles: Product Owner, Team, Scrum Master
  • Product Backlog, Sprint Backlog, few compact reports
  • Short work cycles (-> ”Sprints”) for incremental development

• Based on the Agile Manifesto of Kent Beck at al.
  • Human-centred
  • Technology and tools have secondary role
  • Close cooperation with customer
  • Empirical learning process

Scrum does not define a development methodology, QA strategy, or risk management approach, but asks the team to take care of these issues appropriately.
Scrum Process – Simplified Overview
Scrum: Backlogs

**Product Backlog**
- Collection of requirements (user stories) for the product – at project start
  - a few, little detailed user stories; collection evolves over time and requirements will be refined over time
- Managed by the **Product Owner**

**Sprint Backlog**
- Collection of requirements (user stories) that are selected for implementation during next sprint
- Managed by the **Team**
Scrum: Sprint

Sprint

• Period (1-4 weeks) in which a shippable product increment (executable, tested, and documented) is created by the Team

• Time-boxed
  • i.e., ends exactly at the scheduled time

• At end of Sprint: Product Owner accepts/rejects the final results (i.e., the software)
  Partially completed or incorrect results
  • will not be shipped (no compromise on quality) and
  • go back to the Product Backlog for inclusion in the next Sprint (Backlog)
Scrum: 3 Roles

Product Owner
• Decides which requirements are implemented for a product version
• Decides about when product increments will be shipped

Team
• Implements requirements
• Decides how many requirements are implemented in a Sprint
• Organizes its activities (-> tasks) independently

Scrum Master
• Takes care of the proper implementation of Scrum
• Supports the team in process-related issues
Sprint Meetings

Product Backlog: Emerging, prioritized requirements

Vision: Anticipated ROI, Release, Milestones

Sprint Backlog

Selected Product Backlog

Now functionality is demonstrated at end of sprint

Daily Scrum

Sprint Planning Meeting

Sprint Review Meeting

Sprint Retrospective Meeting

1 day for a 4 week Sprint

15 min daily

~1 hour
Sprint Burndown Chart – Example

Sample Burndown Chart

Remaining effort (person-hours)

Remaining and completed tasks

Completed tasks

Remaining tasks

Ideal burndown

Remaining effort

Remaining tasks

User Story  Task

<table>
<thead>
<tr>
<th>Priority</th>
<th>Requirement</th>
<th>To be Done</th>
<th>In Progress</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sprint Burndown Chart – Example

Exercise:
How many people are in the Scrum team?
Assumptions:
1. Only 70% of the full capacity are planned/allocated on day 0
2. A (work) day has 8 (work) hours

- Completed tasks
- Remaining effort
- Ideal burndown
- Remaining tasks
Sprint Burndown Chart – Example

Answer:
225 ph is ca. 70% of 320 ph
1 person works 8 x 20 = 160 ph per month
Thus, there are probably 2 persons on the team

How many people are in the Scrum team?
Assumptions:
1. Only 70% of the full capacity are planned/allocated on day 0
2. A (work) day has 8 (work) hours
Sprint Burndown Chart – Example

Additional questions:
1. Are there any new tasks added during the sprint?
2. How do we know how much effort was spent?
Sprint Burndown Chart – Example

Additional questions:
1. Are there any new tasks added during the sprint?

Answer:
No, doesn’t look like; whenever a task is done, the number of remaining tasks drops accordingly.
Sprint Burndown Chart – Example

Additional questions:
2. How do we know how much effort was spent?

Answer:
Since we know how many people are on the team (and we don’t allow overtime):
Spent Effort = 2 \times \#Days \times 8 \text{ ph}
Scrum: Prerequisites and Risks/Challenges
Scrum: Prerequisites and Risks

• Scrum has a different perspective on employees, management, distribution of power as compared to traditional project management approaches
  • In particular, higher and top-level management must understand and actively support Scrum

• Customer also must re-think their role
  • Close involvement through many iterations is often unfamiliar
  • Creates additional work on the client side
  • Not every customer wants to see the creation of the product
Scrum: Prerequisites and Risks (cont’d)

• Partitioning of the product
  • Product must be partition-able so that it can actually be developed incrementally
    • For example, this is difficult in certain regulated industries that have to certify full requirements specifications very early
  • Not all requirements can be partitioned equally well
  • In particular, non-functional requirements, such as performance, safety, security are difficult to partition – and must therefore be re-examined in each iteration (integration) and ensured
Scalability of Scrum

- Typical individual team is 7 ± 2 people
  - Scalability comes from teams of teams
- Factors in scaling
  - Type of application
  - Team size
  - Team dispersion
  - Project duration
- Scrum has been used on multiple 500+ person projects (e.g., SAP)
Structure of Lecture 11

- Light-weight processes / Evolutionary development
- Agile Processes/Methods
  - Extreme Programming (XP)
  - Scrum (intro)
- KANBAN
- Lean Processes/Methods
Kanban (Jap.): literally 'signboard' or 'billboard'

**SCRUM** vs. **KANBAN**

- Time-Boxing Velocity
- Task-Boxing Lead-Time
A Scrum team will only commit to items that they think they can complete within one iteration (based on the definition of “Done”). If an item is too big to fit in a sprint, the team and product owner will try to find ways to break it into smaller pieces until it does fit. If items tend to be big, iterations will be longer (although usually no longer than 4 weeks).

Kanban teams try to minimize lead time and level the flow, so that indirectly creates an incentive to break items into relatively small pieces. But there is no explicit rule stating that items must be small enough to fit into a specific time box. On the same board we might have one item that takes 1 month to complete and another item that takes 1 day.
Scrum has sprints (iterations) of 1 (to 4) weeks (= time box) but: it is not always easy to divide the tasks or features of the systems to fit into such time intervals.

What about instead limiting the amount of tasks or features (= task box) that can be worked on concurrently and deliver when finished?
Velocity vs. Lead-time

SCRUM focuses on:
• Flow of work items (throughput/velocity) = the number of features (user stories, tasks, etc.) implemented per unit of time (with given workforce)

KANBAN focuses on:
• Lead-time (cycle time) = the average time it takes to finish a work item (from start to end)
Visualization of Work Flow: Issue Tracker

Example: PivotalTracker (www.pivotaltracker.com)
Visualization of Work Flow: Issue Tracker

Example: PivotalTracker
Visualization of Work Flow: Issue Tracker

Show several projects concurrently
Visualization of Work Flow: Issue Tracker

Analytics
Scrum Board versus Kanban Board

From: Kanban and Scrum - making the most of both by Henrik Kniberg and Mattias Skarin on Dec 21, 2009
Kanban Board

A Work Item represents a unit of work to be carried out by the development team.

Describe a Work item on a post-it sheet and put it on a board in one of the categories: "To do", "Ongoing" or more detailed states. "Done" shows the Work Items that are finished.

From: Kanban and Scrum - making the most of both by Henrik Kniberg and Mattias Skarin on Dec 21, 2009
What is the right WIP limit?

We’re idle & bored! Let’s increase WIP limit to 8!
What is the right WIP limit?

To do  Ongoing  Done :o)

H  I  E

D  L  F

Problem with integration server. Can’t finish D & E! We’ll work on F & G instead!

Oops. WIP limit reached. Now we HAVE to stop and fix the integration server!

Let’s reduce WIP limit to 4, so we react earlier next time!
Differences between Scrum and Kanban

Cannot add items to ongoing iteration.
A sprint backlog is owned by one specific team

Prescribes 3 roles (PO/SM/Team)
A Scrum board is reset between each sprint
Prescribes a prioritized product backlog

Can add new items whenever capacity is available
A Kanban board may be shared by multiple teams or individuals

Doesn’t prescribe any roles
A Kanban board is persistent
Prioritization is optional.
Similarities between Scrum and Kanban

• Both use pull scheduling
• Both limit WIP (but in different ways)
• Both use transparency to drive process improvement
• Both focus on delivering releasable software early and often
• Both are based on self-organizing teams
• Both require breaking the work into pieces
• In both, work flow is continuously optimized based on empirical data (velocity / lead time)

• Both are Lean
More prescriptive

RUP (120+)

<table>
<thead>
<tr>
<th>RUP Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business, User, System, Manage, Plan, Design, Analyze, Build, Test, Deploy, Manage, Control, Support, Improve</td>
</tr>
</tbody>
</table>

More adaptive

Scrum (9)

<table>
<thead>
<tr>
<th>Scrum Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprint, Define, Do, Test, Review, Retrospect</td>
</tr>
</tbody>
</table>

Kanban (3)

<table>
<thead>
<tr>
<th>Kanban Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visualize the value, Limit WIP, Time and flow</td>
</tr>
</tbody>
</table>

Do Whatever (0)

Visualize Your Workflow
Limit Your WIP
Use Lead-Time as default metric

RUP has over 30 roles, over 20 activities, and over 70 artifacts.
Structure of Lecture 11

- Light-weight processes / Evolutionary development
- Agile Processes/Methods
  - Extreme Programming (XP)
  - Scrum (intro)
- KANBAN
- Lean Processes/Methods
Origins of Lean Software Development

- Originates from Toyota Production System (TPS)
  - Also called Just-In-Time system

- Post WWII Japanese automobile industry could not compete with U.S. mass production systems

- Inspiration for TPS found in the 1950’s from U.S. supermarkets
  - Customers could get what they wanted, when they wanted it and shelves were refilled when items were about to run out.

- The concepts transferred to the domain of software engineering by Mary and Tom Poppendieck (2003, 2007).
Main Goals of LEAN

1. **All processes shall give value**
   - Remove everything that does not create value

2. **Ensure good flow in the processes to avoid bottlenecks and queues** (-> work not piling up & waiting)

3. **All activity shall be based on need** (-> Pull)
   - If there is no demand for a product or service, the related task is unnecessary

4. **Become a learning organization with focus on continuous stepwise improvement**
   - Kaizen (= small change for the better)
Focus on reducing the activities that do not create value

Traditional approach

Focus on the efficiency of the activities that create value for the customers

- 60-70% (waste)
- 20-30% (value adding)
- 10-20% (imposed waste)
Focus on reducing the activities that do not create value

The approach to continuous improvement

Focus on removing/reducing the activities that do not create value for our customers

Traditional approach

Focus on the efficiency of the activities that create value for the customers
Seven Principles of Lean SW Development
(by Mary Poppendieck)

1. Optimize the Whole
2. Eliminate Waste
3. Build Quality In
4. Learn Constantly
5. Deliver Fast
6. Engage Everyone
7. Keep Getting Better
Seven Wastes of Software Development

- **Handoffs.** Passing the information/work to someone else, getting information/work from someone else.
- **Partially done work.** Something that is not done. E.g. untested code, undocumented or not maintained code.
- **Task switching.** How many other tasks people need to do. E.g. the amount of projects done simultaneously.
- **Delays.** Waiting for something.
- **Extra features.** Something that is not really needed.
- **Defects.** Something that does not meet the targets, or is not what it is supposed to be. E.g. software bugs, incorrectly implemented business requirements.
- **Relearning (waste of knowledge).** E.g. forgetting decisions, re-trying solutions already tried, the inability to utilize the knowledge of other people.
Industry Guest Lecture

• Date/Time:
  • Friday, 26-Nov, 10:15-12:00
• Topic: “Software Craftsmanship, the Codeborne Way”
• Speaker: Anton Keks, Codeborne (Co-founder, CEO)

• For you to do:
  • Work on homework assignment 6
  • Go to assessment lab